VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Vizualizace konečných prvků
Finite Elements Visualization

2012 Jan Vašata

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Jan Vašata**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Vizualizace konečných prvků
Finite Elements Visualization

Zásady pro vypracování:

Cílem práce je implementace pokročilých vizualizačních doplňku do aplikace pro zobrazování těles metodou konečných prvků. Bude se jednat zejména o techniky vyhlazování těles, vizualizací na povrchu tělesa, pokročilé osvětlovací modely, dekompizice těles na jejich podčásti, a další. Dále se bude jednat o odpovídající úpravy uživatelského rozhraní.
1. Představení hlavních postupů a metod pro vizualizaci konečných prvků.
2. Analýza a návrh vybraných vizualizačních modulů.
3. Implementace.
4. Výkonnostní testy a jejich vyhodnocení.

Seznam doporučené odborné literatury:

Edward Angel: Interactive Computer Graphics, ISBN-10:032153586 (2009)

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Petr Gajdoš, Ph.D.**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Souhlasím se zveřejněním této diplomové práce dle požadavků čl. 26, odst. 9 *Studijního a zkušebního řádu pro studium v magisterských programech VŠB-TU Ostrava.*

V Ostravě 30. dubna 2012 .....

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 30. dubna 2012 .....

Rád bych na tomhle místě poděkoval svému vedoucímu Ing. Petru Gajdošovi Ph.D. za přátelský přístup a pomoc při řešení těžkých úloh.

Abstrakt

Cílem diplomové práce je implementace pokročilých vizualizačních doplňků do aplikace pro zobrazování těles metodou konečných prvků. Zejména se jedná o techniky vyhlazování těles, pokročilé osvětlovací modely, dekompizice těles na jejich podčástí, a další. Mezi vedlejší úlohy patří například úpravy uživatelského rozhraní. Obsahem jsou i výkonnostní testy a jejich vyhodnocení.

Klíčová slova: OpenGL, QT, vizualizace konečných prvků, dekompozice, vyhlazování

Abstract

The main aim of this thesis is the implementation of the advanced visualization plug-in/addition to the application for viewing finite element method bodies. In particular, it is the techniques of body smoothing, advanced shading models, decomposition bodies on their subbodies etc. One of the secondary tasks is for example editing user interface. The thesis also includes performance tests and their evaluation.

Keywords: OpenGL, QT, finite elements visualization, smooth

Seznam použitých zkratek a symbolů

DP	–	diplomová práce
FPS	–	frame per second
CPU	–	central processing unit

Obsah

1 Úvod **5**

2 Návrh a analýza **6**
2.1 Data 6
2.2 Dekompozice 8
2.3 Vedlejší úlohy 11

3 Metody vizualizace **14**
3.1 Reprezentace těles 14
3.2 Zobrazovací metody 17

4 Implementace **23**
4.1 Qt 23
4.2 Struktura aplikace 23
4.3 Implementované algoritmy 24

5 Testování **36**
5.1 Testování - fps 38
5.2 Testování - CPU 40
5.3 Testování - Pamět 40

6 Závěr **42**

7 Reference **43**

Seznam tabulek

Seznam obrázků

1 Geometrická tělesa . . . 6
2 Hexahedron . . . 7
3 Dekompozice Tree . . . 9
4 Příklad výpočtu hran pomocí zametací roviny. . . . 10
5 Tuhost pružin . . . 10
6 Průměrování normál . . . 12
7 Ukázka phong shaderu z aplikace . . . 13
8 Stavba hraničního modelu . . . 14
9 Trojúhelník . . . 15
10 CSG . . . 16
11 Mřížka . . . 16
12 Raytracing . . . 19
13 Bounding box . . . 19
14 Hiearchie bounding boxu . . . 20
15 Octree . . . 20
16 Ukázka Raytracingu . . . 21
17 Schéma vypočtu radiosity . . . 21
18 Ukázka dekompozice . . . 24
19 Ukázka vyhlazení . . . 34
20 Ukázka QtTreePropertyBrowser . . . 35
21 Ukázka model 1 . . . 36
22 Ukázka model 2 . . . 37
23 Ukázka model 3 . . . 37
24 Testování fps - stav v klidu . . . 38
25 Testování fps -pracovní stav . . . 38
26 Testování fps s průhledností . . . 39
27 Testování CPU . . . 39
28 Testování paměti . . . 40
29 Testování paměti/velikost modelu . . . 41

Seznam výpisů zdrojového kódu

1 Algoritmus pro rozklad Simple (UpdateMatrix) 25
2 Pseudo algoritmus pro rozklad Tree (CreateLogic) 26
3 Pseudo algoritmus pro rozklad Tree (CallUroven) 27
4 Pseudo algoritmus pro rozklad Tree (NajdiShlukyRodic) 27
5 Pseudo algoritmus pro rozklad Tree (UpdateMatrix) 28
6 Pseudo algoritmus pro rozklad Springs (CreateLogic) 1/4 28
7 Pseudo algoritmus pro rozklad Springs (CreateLogic)2/4 29
8 Pseudo algoritmus pro rozklad Springs (CreateLogic)3/4 30
9 Pseudo algoritmus pro rozklad Springs (CreateLogic)4/4 31
10 Pseudo algoritmus pro rozklad Springs (UpdateMatrix) 31
11 Pseudo algoritmus pro otočení normály 33
12 Pixel shader 33

1 Úvod

Cílem diplomové práce je implementace pokročilých vizualizačních technik pro tělesa metodou konečných prvků. Hlavní náplní je implementace algoritmu pro vyhlazování, osvětlování a dekompozici těles. Vedlejší úlohou je návrh přehledného a jednoduchého grafického uživatelského rozhraní.

Vizualizace dat je obecně dobrý nástroj pro určitý pohled na takovou skupinu informací, která by v textové podobě byla velká a nepřehledná. Vizualizací existuje mnoho typů, jedním z nich je i FEM.

FEM je numerická metoda řešení, která se snaží poskytnout přibližné řešení. Tato metoda je široce využívaná při řešení statických i dynamických problémů v různých odvětvích, jako jsou například mechanika kapalin, biomedicina, elektromagnetických jevů apod. Hlavním principem této metody je diskretizace spojitého prostoru na konečné množství prvků a následné interpolace mezi důležitými parametry, které jsou ve vrcholech těchto prvků.

Tato práce si mimo základních cílů, které byly vyjmenovány výše, klade i další cíl, a to zachování nebo vylepšení určitých vlastností aplikace, ze které vychází. Mezi tyto vlastnosti patří hlavně rychlost vykreslení výsledků a využití paměti RAM.

Tato diplomová práce navazuje na již existující projekt Ing. Petra Gajdoše, Ph.D. a doplňuje ho o výše uvedené funkce.

Hlavní struktura práce je:

1. Představení hlavních postupů a metod pro vizualizaci konečných prvků.
2. Analýza a návrh vybraných vizualizačních modulů.
3. Implementace.
4. Výkonnostní testy a jejich vyhodnocení.

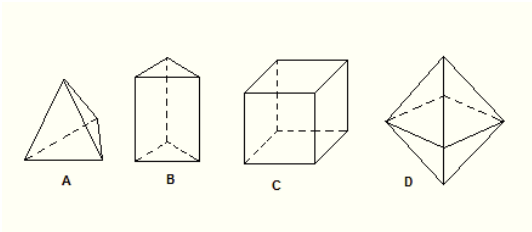
2 Návrh a analýza

V této kapitole jsou informace o náhledu na strukturu modelu, která je zobrazována v aplikaci. Nalezneme zde základní pohled na reprezentaci modelu, taktéž jsou zde popsány zdroje dat pro modely, se kterými dokáže aplikace zacházet. V poslední části této kapitoly nalezneme popis způsobu uložení dat v paměti počítače.

2.1 Data

2.1.1 Reprezentace modelu

Objekt, který je v aplikaci vykreslován a na kterém se provádí výpočty, je tvořen modelem skládající se ze základních elementárních částí. Model se skládá z domén, které slouží k rozdělení složitých modelů na menší části. Nad doménami se provádí operace posun, rotace nebo rozklad. Doména již nejde dále dělit, jedinou výjimkou je řezaní pomocí ořezové roviny nebo schovávání vybraných elementů.



Obrázek 1: Geometrická tělesa

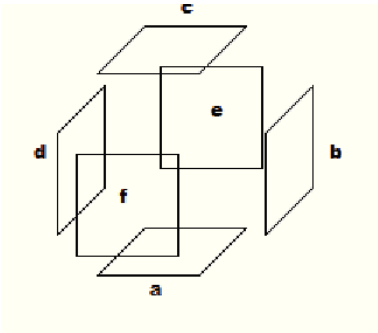
Každá doména se skládá z geometrií. Geometrie jsou geometrické objekty, které mohou nabývat tvaru tetrahedron, pentahedron, hexahedron a octahedron. Základní geometrické objekty jsou vyobrazeny na obrázku 1, kde objekt A je tetrahedron, B je pentahedron, C je hexahedron a D je octahedron.

Sousedící strany jednotlivých geometrií musí na sebe přesně přiléhat. Není možné, aby nějaká strana geometrií měla dva sousedy. Každá geometrie se pak skládá z elementů. Elementy jsou jednoduché základní objekty, jako je trojúhelník a čtverec. Na obrázku 2 je ukázaná stavba hexahedronu. Hexahedron se skládá z 6 obdélníku. Každý element se pak skládá z tří případně čtyř vrcholů a v každém vrcholu nebo na ploše je normála, která se používá pro osvětlování při vykreslování.

Každý model může obsahovat ještě dodatečné informace. Tyto informace se váží k vrcholům geometrií a mají informační charakter, který třeba slouží k výpisu naměřených hodnot teplot nebo hustoty.

2.1.2 Zdroje dat

Data lze nahrávat z lokálního disku nebo z online databáze. Nahrávání z disku podporuje formáty .geo a speciální formát. Pro nahrávání z online databáze je potřeba být přihlášen na školní síti VŠB-TUO.



Obrázek 2: Hexahedron

Formát geo Formát .geo se skládá z jednoho souboru, ve kterém jsou obsaženy všechna data o modelu. Nejsou zde však obsažené dodatečné informace. Strukturu formátu .geo obsahuje:

- název modelu
- datum vytvoření modelu
- nody
- elementy
- vrcholy
- aj..

Speciální formát Skládá se ze 2 textových souboru p.txt, t.txt. Neobsahuje dodatečné informace. V souboru p.txt se nachází seznam vrcholu. Soubor t.txt je seznam elementů.

2.1.3 Struktůra dat

Pro uložení dat v paměti byla zachována struktura modelu, ve kterém je načítána (pentahedron, hexahedron, octahedron,...). Pro rychlejší počítaní některých problémů byla struktura lehce upravena. Přilehlé elementy dvou geometrií byly nahrazeny jedním jediným elementem a znalostí geometrií, ke kterým patří.

Takováto struktura umožňuje rychlejší výpočet viditelnosti okrajových ploch. A při klasickém zobrazení zobrazovat pouze tyto plochy. Vlastností teto struktury je jedna normála pro dvě strany. Nedá se tedy určit, na kterou stranu je element natočený. Tato vlastnost bude ještě připomenuta při popisu implementace osvětlení.

Byla zachována i struktura domén, kdy každá geometrie spadá do nějaké domény. Domény slouží pouze k rozdělení složitého modelu, například pro rozpad či jinou manipulaci. Nakonec jsou všechny elementy přerozděleny do seznamu. Každý seznam má svůj příznak, podle kterého se určuje, kam mají být elementy dány a také se podle flagu řídí vykreslování. Flag obsahuje:

- `GFT_LINE` - Zda se jedná o úsečku.
- `GFT_TRIANGLE` - Zda se jedná o trojúhelník.
- `GFT_QUAD` - Zda se jedná o čtverec.
- `GFM_NONE` - Žádný flag není nastaven.
- `GFM_IS_SELECTED` - Určuje, zda je daný element vybrán.
- `GFM_IS_BOUNDARY` - Určuje, zda je daný element hraničním.
- `GFM_IS_REDUNDANT` - Slouží pro prvotní výpočet. Dále v programu není použit.

2.2 Dekompozice

2.2.1 Úloha dekompozice

U velkých a složitých modelů bývá potřeba zobrazit jejich části tak, aby byly lépe vidět a nespokojíme se s pouhým schováním některých části. Proto používáme dekompozici, tedy obvyklé rozložení části modelu, tak aby každá část byla lépe vidět. Povětšinou se to provádí posunutím nebo rotací části modelu.

Dekompozici provádíme na doménách modelu, protože rozdělení modelu na čisté elementy by bylo špatně realizovatelné. Z těchto domén si bereme pouze jejich pozici (centrum) a pracujeme pouze s ní, jako s bodem.

2.2.2 Typy dekompozice

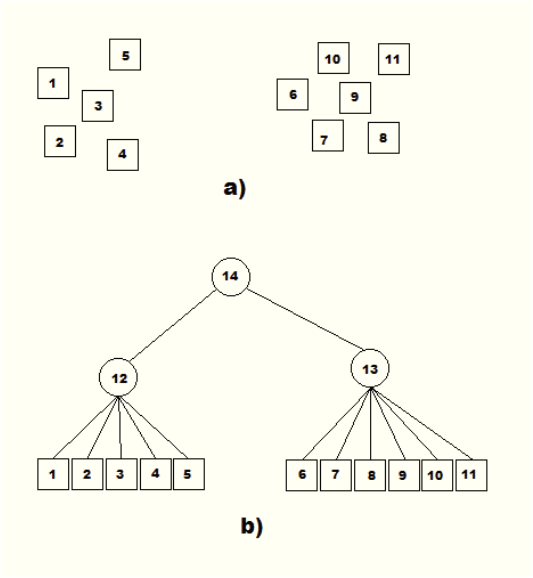
Byly navrhnuty tři typy technik pro dekompozici, které se liší svou komplexností. Od nejjednodušší, která je jednoduchá na ovládání, ale nenabízí prostor k manipulaci, až po komplexní, která je vhodná na složité typy modelu nebo specifické ovládání rozpadu.

Simple Jedná se o jednoduchou dekompozici, kde se jednotlivé domény transformuji od počátku stejně rychle a není možnost je jakkoli ovlivnit či zasahovat do části modelu. Jediné možné ovládání je rychlost rozpadu všech částí po osách x, y a z.

Tree Dekompozice Tree je oproti dekompozici Simple rozšířena o jednoduchou logiku sdružování domén do shluků a kontrolování těchto shluků.

Algoritmus se skládá ze tří kroků. Z počátečních domén se vytvoří shluky a z těchto shluků se vytvoří další shluky. V třetí vlně se z již vytvořených shluků vytvoří jeden velký shluk. Dostáváme tak strom o hloubce tři, kde listy stromu jsou domény objektu. A uzly stromu reprezentují shluky.

Shluky se provádí na jednoduchém algoritmu, který počítá vzdáleností mezi doménami. Pokud je nalezen potencionální shluk, ve kterém je nejdelší vzdálenost mezi doménami menší než předem stanovena hranice, a pokud je počet domén v tomto potencionálním shluku vetší než požadovaná hodnota, tak se z těchto domén vytvoří shluk.



Obrázek 3: Dekompozice Tree

Obdobně tento algoritmus funguje i ve druhém kroku, kdy z vypočtených shluků tvoříme další shluky. Pro výpočet pozice u shluku se používá aritmetický průměr všech pozic domén ve shluku. Na obrázku 3a lze vidět počáteční scénu s 11 doménami a obrázek 3b ukazuje, jak vypadá logické spojení domén ve stromě pomocí shluku po výpočtu.

Následná manipulace se stromem probíhá tak, že si uživatel může označit uzly nebo listy stromu, u kterých chce, aby se na ně dekompozice projevila nebo ne. Poté může použít rozpad po osách x, y a z, který se projeví jen na ty domény, které si uživatel zvolil.

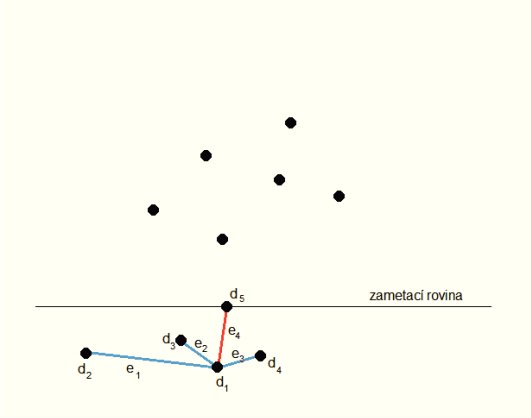
Byly prováděny pokusy o nalezení jiných geometrických útvarů, než je shluk, přesněji kružnice, přímka a rovina. Bohužel od těchto technik bylo upuštěno, protože nedávaly dobré výsledky.

Springs Předcházející dekompozice stavěly své domény do jednoduchých stromů. Což však nemusí vyhovovat u některých složitých modelů. Proto byla navrhnuta třetí dekompozice, která staví domény do grafu. V ideálním stavu je každá doména spojena s nejbližšími doménami hranami v grafu.

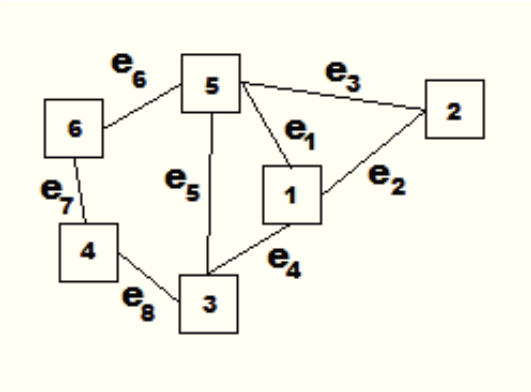
Graf je vytvořen pomocí techniky zametání roviny. Na začátku mějme dva seznamy. V jednom seznamu mějme seřazeny všechny domény v nějaké ose. Druhý seznam bude sloužit pro uchovávání právě zpracovávaných domén. Seznam bude umět uložit odkaz na doménu a k této doméně další tři odkazy na domény, které budou reprezentovat dosud nalezené tři nejbližší domény. Označme tento seznam jako P.

Následně procházíme se zametací rovinou prvním seznamem seřazených domén. U každé domény se zastavíme a provedeme tři kroky.

V prvním kroku zjistíme, zda se v seznamu P nenachází domény, u kterých už můžeme říct, že jsme našli tři nejbližší domény. Pokud doména má už nějaké tři nejbližší domény



Obrázek 4: Příklad výpočtu hran pomocí zametací roviny.



Obrázek 5: Tuhost pružin

vyplněné a vzdálenost od domény k zametací rovině je delší než vzdálenost domény a třetí nejbližší domény, tak jsme již pro danou doménu nalezli 3 nejbližší domény a můžeme ji vyřadit a vytvořit hrany.

V druhém kroku spočítáme pro každou doménu v seznamu P vzdálenost s doménou na zametací rovině a za předpokladu, že je aspoň třetí nejbližší , ji vložíme do seznamu nejbližších domén.

V třetím kroku přidáme do seznamu P doménu ze zametací roviny.

Nakonec, až projdeme celý seznam seřazených domén, tak vytvoříme pro zbývající body seznamu P hrany s dosud nalezenými nejbližšími doménami.

Takto vytvořený graf ovládáme tak, že vybereme doménu, kterou chceme posouvat, a pomocí rozkladu ji posouváme po osách x, y a z. Domény, jenž jsou na tuto doménu připojené , se posouvají o určitou konstantu pomaleji. Další domény, které jsou připojeny k doméně, která se posouvala, se posouvají zase o určitou konstantu pomaleji, atd. Posun jednotlivých domén je znázorněn vzorcem. Domény jsou brány vždy s nejkratší vzdálenosti od vybrané domény.

$$M_k = K^i * P_k$$

Kde:

K - konstanta zpomalení
P_k - rychlost posunu předešlé domény
i - počet uzlů na nejkratší cestě od domény k hlavní doméně

Tuhost pružin (hran) mezi jednotlivými doménami je konstantní a není nijak ovlivněna jejich vzdáleností. Na obrázku 5 lze vidět, že objekty 2, 3 a 5 se budou pohybovat o K původní síly a objekty 4 a 6 o K*K původní síly.

Pro lepší objasnění uveďme ukázku. Na obrázku 4 je situace, kdy v seznamu P je již bod d1, d2, d3, d4. Na zametací rovině je právě počítaný bod d5. Bod d1 má ve svém seznamu nejbližších domén hrany e1, e2, e3. Následně se spočítá vzdálenost d1 a d5 a zjistí se, že hrana e4 je blíže než e1, a proto se nahradí.

2.3 Vedlejší úlohy

Při tvorbě diplomové práce vznikaly malé, ale důležité vedlejší úlohy, které bylo potřeba řešit.

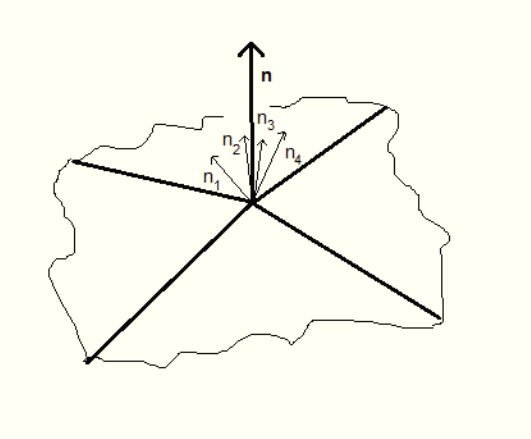
2.3.1 Vyhlazování

Pro lepší vizuální vjem může vzniknout požadavek na vyhlazení povrchu modelu. Pro výpočet algoritmu na vyhlazení povrchu musíme sjednotit normály vrcholů, jenž jsou identické, jenom každý patří jiné geometrii. Při sjednocení normál započítáme nejdřív průměrnou normálu a poté každé normále přiřadíme tuto průměrnou normálu. Ne všechny normály se ale průměrují, obvykle se určí úhel mezi dvěma normálami, který určuje mez, při které daná normála se zprůměruje nebo se vynechá. O zbytek se postarají shadery. Ne každý shader však umí interpolovat normály, záleží, jak je daný shader napsán. Na obrázku 6, lze vidět, jak se ze 4 vrcholu na sobě ležících zprůměrují normály.

Toto však není jediná metoda pro „narovnání normál“, další metodou může být přiřazení váhy každé normále, podle toho, jak je daný polygon, jehož je normála vrcholu součástí, velký a poté, při rovnání normal se normály s vetší váhou podílejí více než normály s menší váhou.

2.3.2 Shadery

Grafické karty umožňují programovat část grafického řetězce grafické karty. Takovýmto programům říkáme shadery. Shadery se programují pomocí speciálních programovacích jazyků, např. GLSL, Cg, HLSL. V dnešní době máme několik typu shaderu. Mezi základní patří vertex shader, pixel shader a geometri shader. Vertex shader - Na vstupu přebírá jeden vrchol a na výstupu vrací jeden vrchol. Uvnitř funkce se obvykle realizuje nějaká transformace daného vrcholu. Geometry shader - Oproti vertex shader umí generovat nebo odebírat vrcholy. Tento shader můžeme použít třeba ke generování vegetace. Pixel shader - Používá se u rasterizace a určuje výslednou barvu pixelu.



Obrázek 6: Průměrování normál

Tato diplomová práce používá pouze vertex a pixel shader. Pro více informací o shaderech doporučuji přečíst [2],[3],[4],[5] a [6].

Phongův osvětlovací model Phonguv osvětlovací model na rozdíl od Gauradovoho stínovaní interpoluje normály namísto pouhé interpolace barvy, a tím dosahuje lepší vizuální vjem, hlavně spekulární složka je počítána lépe. Při výpočtu se do sebe započítává difusní, ambientní a spekulární složka.

Difusní složka Difuzní složka nám udává, jak se odráží světlo od daného materiálu.

$$I_d = C_d * (I * n) * L_d \tag{1}$$

Ambientní složka Udává, jak moc daný materiál emituje vlastní světlo.

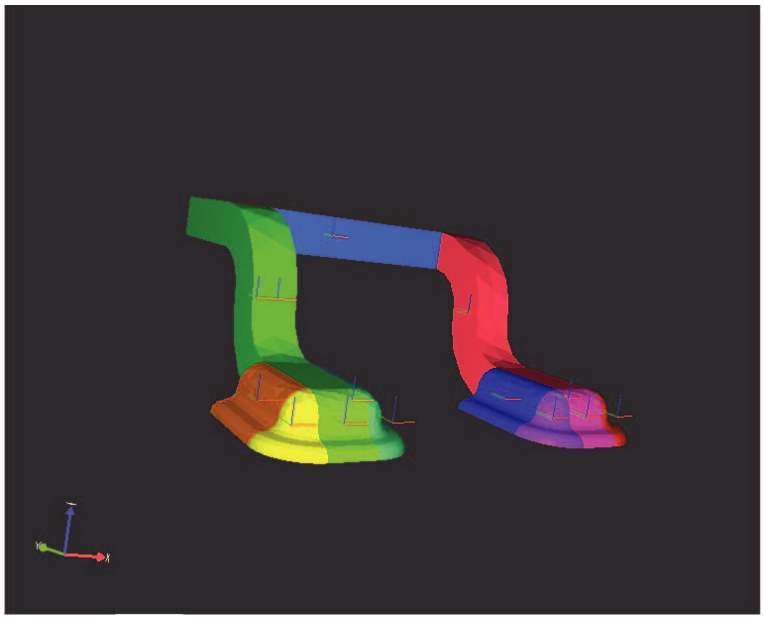
$$I_a = C_a * L_a \tag{2}$$

Spekulární složka Spekulární složka je zde pro výpočet odlesku materiálu.

$$I_s = C_s * L_s * cos^{\alpha}(\gamma) \tag{3}$$

Celková intenzita se počítá pro každou složku RBG zvlášť.

$$I = I_a + I_d + I_s \tag{4}$$



Obrázek 7: Ukázka phong shaderu z aplikace

3 Metody vizualizace

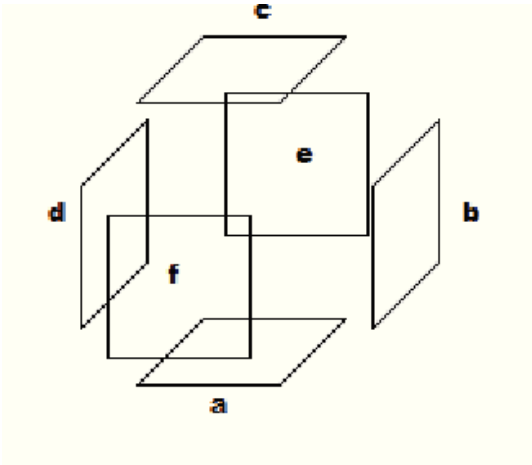
V této kapitole jsou popsány zakladní typy dat. Probírány jsou hraniční reprezentace přes CGS až po voxelovou reprezentaci. Následně jsou popsány techniky pro zobrazování hraničních těles a objemových těles. U hraničních těles to je hlavně základní zobrazovací řetězec, ray tracing a radiozita.

3.1 Reprezentace těles

3.1.1 Hraniční reprezentace

Nejběžnější reprezentaci objektu, se kterou se můžeme setkat, je hraniční reprezentace. Objekt je reprezentován pouze obalem. Pro zobrazení není potřeba vědět, co je uvnitř objektu, a tak se s informací o vnitřní reprezentaci objektu nepočítá. Obal nesmí být nikde přerušen, musíme být schopni rozlišit mezi bodem, který by byl uvnitř nebo venku modelu.

Na obrázku 8 vidíme, jak vypadá například hraniční model krychle.

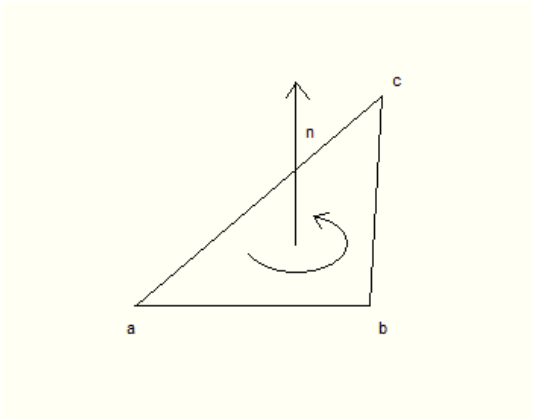


Obrázek 8: Stavba hraničního modelu

Nejčastěji se takovýto model skládá z bodů, hran a stěn. Pokud by měla být vyobrazena nějaká zakřivená plocha, tak se aproximuje na trojúhelníky. Další potřebnou informací je normála bodu a stěny. Tato informace není potřebná pro konstrukci modelu, ale neobejdeme se bez ní při vykreslení modelu. Hraje nedílnou součást v pokročilých osvětlovacích modelech jako je Phongův osvětlovací model.

Nejstarší hraniční reprezentaci je hranová reprezentace. Skládá se pouze ze seznamu bodů a seznamu hran. Výhodou hranové reprezentace je její úspora, avšak nevýhodou je nejednoznačnost objektu. Díky chybějící informaci o plochách může nastat případ, kdy má více různých reprezentací těles.

Rozšířením hranové reprezentace o informaci s plochami dostaneme ploškovou reprezentaci. Seznam hran nahradíme seznamem ploch. Nejčastěji se body plochy zapisuji



Obrázek 9: Trojúhelník

proti směru hodinových ručiček, abychom byli schopni odvodit správnou normálu plochy a nemuseli tak uchovávat tuto informaci zvlášť.

Nejzákladnější ploškou je použit trojúhelník, protože má dobré vlastnosti, například všechny jeho body jsou vždy v rovině, je nejmenší možnou plochou a může se dobře dělit.

Bodová reprezentace je speciálním případem. Uchováváme pouze body na povrchu tělesa. K bodu obvykle přidáváme nějakou dodatečnou informaci, například barvu. Obecně ale těchto bodů jsou miliony (až stovky GB) a tedy tato metoda má velkou paměťovou náročnost. Bodovou reprezentaci nejčastěji najdeme u skenování objektů.

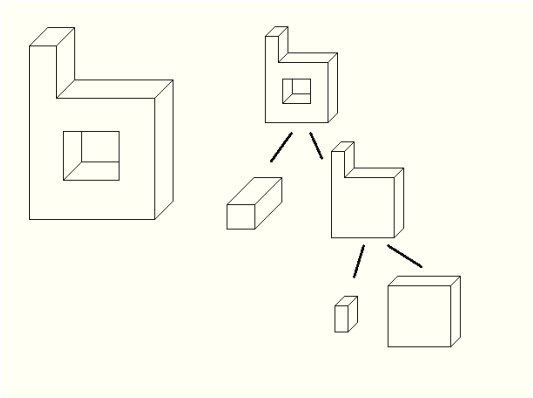
Okřídlené plošky Tato reprezentace uchovává u hran informaci o bodech na začátku a konci, o všech přilehlých hranách, sousedních plochách a libovolnou další hranu. Výhody: rychlé nalezení sousedních ploch, rychlé nalezení smyčky hran, plocha nemusí být konvexní, plocha může obsahovat díry.

Konstruktivní geometrie těles Z jednoduchých geometrických těles jako je kvádr, válec, jehlan, ale i NURBs plocha, vytvoříme pomocí množinových operací výsledné těleso. Mezi množinové operace řadíme průnik, sjednocení a rozdíl. Operace a jednoduchá tělesa řadíme do stromu, tak abychom dostali očekávané komplexní těleso.

3.1.2 Objemová reprezentace

Pro vizualizaci mraku nebo výsledku z tomografu se hraniční reprezentace nehodí. Proto se používá objemová reprezentace. Data se většinou uchovávají v mřížce. Mřížka může nabývat 7 různých podob:

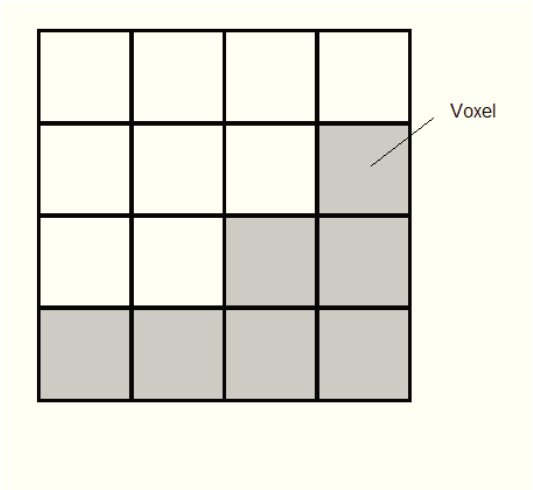
- kartézská
- pravidelná



Obrázek 10: CSG

- pravoúhlá
- strukturovaná
- nestrukturovaná
- blokově strukturovaná
- hybridní

Nejčastěji se setkáme s pravidelnými prostorovými mřížkami. Hodnotou v mřížce je základním elementem voxel, který nabývá nejčastěji tvaru kvádru. Má v celém svém objemu stejnou hodnotu. Je to analogicky protějšek k pixelu v 2D grafice. Pro více informací o objemové reprezentaci těles doporučuji přečíst [27].



Obrázek 11: Mřížka

Pro výpočet objemových dat, například z CT snímku, můžeme použít Marge Cube. Jde o metodu, kdy jsou v paměti načteny vždy jen dva sousední snímky, které se zpracovávají po krychlích.

Pro více informací o této problematice doporučuji přečíst [14],[15],[16],[17] a [18].

3.2 Zobrazovací metody

Zobrazovací metody můžeme rozdělit na dva typy. Ty, které se snaží o rychlé vykreslení a na ty, které se snaží o kvalitnější (realističtější) vykreslení. Bohužel, nemůžeme dosáhnout oboje. Vždy je potřeba se rozhodnout, kterou strategii zvolíme, například pro modelovací editor budeme chtít rychle vykreslení, ideálně 60 snímků za sekundu. Naopak, pro výsledný render celovečerního animovaného filmu budeme chtít výsledek kvalitní, na hranici realističnosti. Pro více informací o zobrazovacích metodách doporučuji přečíst [21],[22],[23],[24] a [25].

3.2.1 Standartní zobrazovací řetězec

Jedná se o metodu, která má za cíl vykreslit obraz rychle na úkor realističnosti. Je to základní stavební kámen, například pro CAD programy, počítačové hry nebo modelovací editor Blender. Řetězec se skládá z 8 kroků, které se však můžou odlišovat podle konkrétní situace, například podle toho, jaký zvolíme osvětlovací model (Phong, atd.).

Na začátku musíme mít popis scény, kterou chceme vykreslit. Obsahuje množinu objektů, které chceme vykreslit. Dále množinu světel, které se ve scéně nachází, a popis, jak danou scénu chceme vykreslit (místo, odkud se chceme dívat, atd...).

Prvním krokem je vygenerování ploch (nejlépe vygenerování trojúhelníků) ze všech objektů ve scéně, které aproximují jejich povrch. Pokud se již naše těleso skládá z těchto ploch, tak můžeme tento krok přeskočit, pokud však je naše těleso popsáno jako koule o průměru r, tak je třeba zvolit dobrou aproximační techniku, která udělá z koule množinu trojúhelníků s body a normálami ve vrcholech.

Druhým krokem je odstranění těch ploch, které zcela jisté nepůjdou vidět. Každý trojúhelník ve scéně má svou vlastní normálu, tato normála nám určuje směr natočení trojúhelníka. Pokud je normála ve stejném směru jako pohled, kterým se na scénu díváme. Tak lze předpokládat, že daný trojúhelník se nachází na protější straně objektu, a tudíž nebude vidět. Odtud tedy můžeme zjistit, které trojúhelníky v tomto kroku můžeme vyřadit. K výpočtu použijeme skalární součin normály plochy a směru našeho vykreslovacího pohledu.

Třetím krokem je výpočet osvětlení ve vrcholech každé plochy. Osvětlení je jedním z důležitých kroků, protože určuje, jak budeme vnímat dané těleso. Mezi nejjednodušší techniky řadíme konstantní stínování, které jednotlivé plošky vyplní stejnou barvou. Pokročilejší je Gouradovo stínovaní, které vypočte v každém vrcholu vlastní barvu, a poté ji interpoluje po celé plošce. Rozšířením je Phongovo stínování, které interpoluje normály z vrcholu, a až poté vypočítává barvu daného pixelu. Osvětlení se ovšem netýká vrhání stínu. Na stíny se používají jiné techniky.

Čtvrtým krokem je provedení transformace pro všechny vrcholy. Předem vypočtenou promítací matici vynásobíme s vrcholem v homogenních souřadnicích.

Pátým krokem je ořezání zorným objemem. Zde se odstraní další plochy, které nepůjdou vidět. Díky tomu se nebudou muset kreslit plochy, které náš pohled nezabírá, a dojde tím ke zrychlení vykreslovacího řetězce.

Šestým krokem je přechod od homogenních souřadnic k afinním souřadnicím.

Sedmým krokem je transformace souřadnic vrcholu na souřadnice výstupního zařízení.

Posledním osmým krokem je rasterizace. Pro viditelnost se používá Z-buffer.

3.2.2 Monte Carlo metody

Monte carlo je stochastická metoda. Využívá pseudonáhodné číslo, které aproximuje řešení. Výhodou těchto metod je, že používají tzv. sledování paprsku. Lze v nich snadno vypočítat zrcadlový obraz, empirická složitost je log (n) a mají nízkou paměťovou náročnost.

Rekurzivní sledování paprsku Z počátku byly metody založené na sledování paprsku od zdroje světla. Tyto metody jsou založené na sledování paprsku od zdroje světla k pozorovateli. Bohužel, mnoho paprsku od zdroje nedorazí k pozorovateli, a tedy počítají se zbytečně. Proto se výpočet obrátil, a počítají se paprsky od pozorovatele ke zdroji světla, takovéto metodě říkáme rekurzivní sledování paprsku. Výhodou této techniky je automatické počítání světelných odrazů a stínů. Také je implementačně velmi jednoduchá.

Nevýhodou je práce pouze s bodovými světly a tudíž stíny, které tvoří, jsou ostré. Ray tracing, jak se také někdy tato metoda nazývá, poskytuje kvalitní výsledky oproti standartnímu zobrazovacímu řetězci.

Algoritmus pro výpočet je velmi snadný. Od pozorovatele vyšleme paprsek skrz počítaný pixel. Na tomto paprsku najdeme nejbližší průsečík s objektem ve scéně. Pokud není žádný průsečík, vrátíme barvu pozadí. V opačném případě se na nalezeném průsečíku vypočítá barva. Barva se počítá ze samotného objektu a z příspěvků od okolních světel. Následně se vypočítá paprsek odrazu, a pokud je objekt průsvitný, tak i lomený paprsek a celé se to rekurzivně opakuje. Počet odražení jednoho paprsku se obvykle omezuje, protože čím vícekrát se paprsek odráží, tím méně daný odraz přispívá do výsledného pixelu.

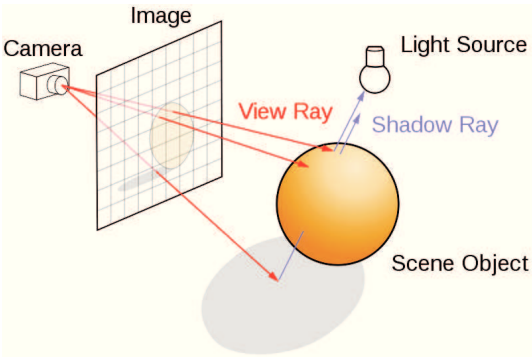
Na nalezeném průsečíku se počítá barva podle vzorce.

$$I(P) = I_L(P) + I_G(P) = I_L(P) + K_n o I(P_o) + K_n p I(P_p)$$

Kde:

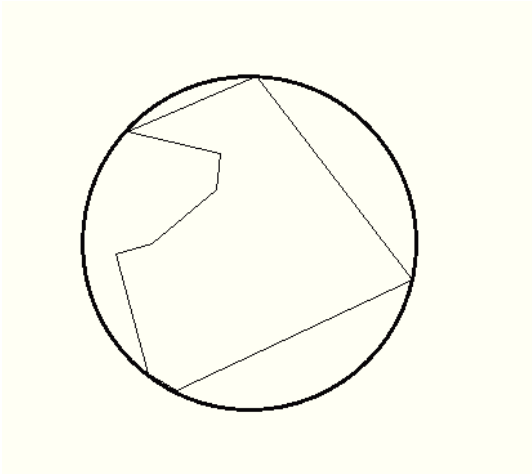
P - průsečík
P_o - další průsečík odraženého paprsku
P_p - další průsečík lomeného paprsku
K_{no} - koeficient odrazu
K_{np} - koeficient lomu

Metoda rekurzivní sledování paprsků je náročná na výkon. Proto se výpočet snažíme zefektivnit. Tělesa uzavíráme do jednoduchých bounding boxu. Příkladem může být ku-



Obrázek 12: Raytracing

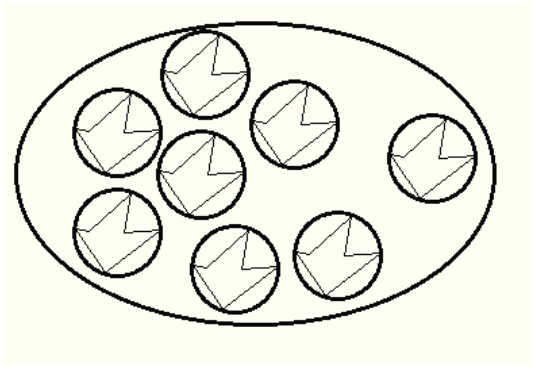
lová plocha, která obepíná celé těleso co nejtěsněji. Při hledaní nyní nejdříve provedeme výpočet paprsku s bounding boxem a pokud se nalezne průsečík, tak až teprve potom děláme složitější průsečík paprsku s tělesem.



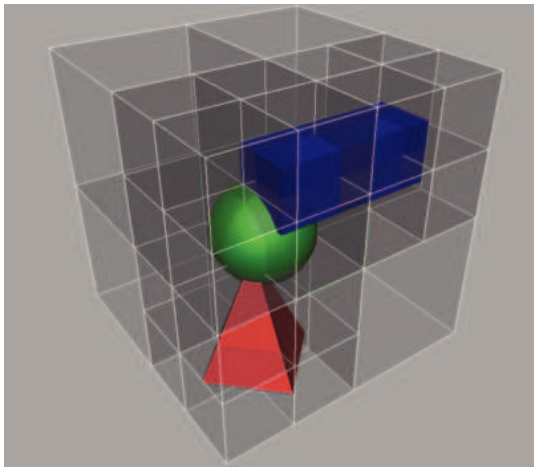
Obrázek 13: Bounding box

Techniku můžeme zdokonalit tím, že do boundin boxu uzavřeme více těles, které jsou blízko u sebe. Můžeme takto rekurzivně vytvořit více bounding boxu a vytvořit strom, ve kterém bude hledaní průsečíku snažší.

Jiná metoda používá pro rychlejší hledání průsečíků Octree. Jde o kvadrantový strom, který prostor rekurzivně dělí na 8 dalších podprostoru. Každý podprostor je tvaru krychle se seznamem těles, které do toho podprostoru spadají. Dělení je ukončeno, pokud je prostor prázdný, nebo se tam nachází jen jedno těleso, nebo dělení prostoru překročilo určitou mez. Následné skládání průsečíku se nejdřív provádí s hledáním průsečíku v podprostoru, a až pokud je nalezen průsečík s podprostorem, je hledán průsečík s tělesem v tomto podprostoru. Díky výhodám stromu je vyhledání průsečíku zrychleno.



Obrázek 14: Hiearchie bounding boxu



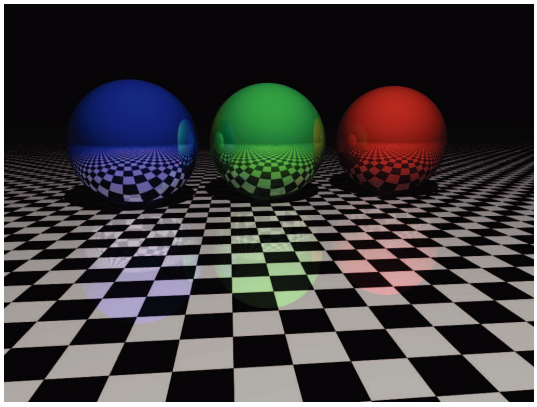
Obrázek 15: Octree

Jednou z nevýhod je i vznik aliasingu. Paprsek se obvykle vysílá středem pixelu, to může mít za příčinu vzniku nehezkých zubů, například na okraji koule. Aliasingu se můžeme zbavit vysíláním více paprsku v různých bodech pixelu a výsledný pixel vytvořit zprůměrováním vyslaných paprsků. Nebo můžeme použít metodu třesení paprsku, kdy se vysílá jen jeden paprsek, ale vysílá se náhodným místem pixelu.

3.2.3 Radiozita

Radiozita je jedna z dalších metod pro vykreslení scény. Oproti sledování paprsku však její výpočet nezávisí na pozici pozorovatele. Stačí ji pro scénu jednou započítat a poté je možné rychlé vykreslení v jakékoliv pozici pozorovatele. Tato metoda se nejlépe hodí na interiéry. Nezvládá však vykreslovat zrcadlový odraz.

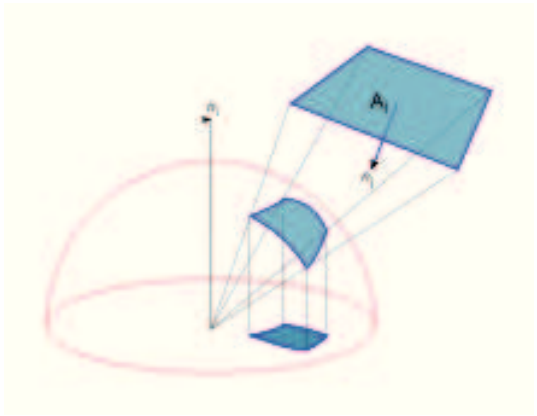
Celá scéna je pokryta ploškami. Každá ploška reprezentuje, jak je daná část scény osvícena. V oblastech, kde nedochází k žádným velkým změnám, jsou plošky velké, naopak v oblastech, kde dochází k velkým změnám osvětlení (okraje stínu, apod...), jsou



Obrázek 16: Ukázka Raytracingu

malé. Na začátku je rozdělení plošek ve scéně uděláno intuitivně, v dalších krocích je ale dělení upraveno, tak aby lépe vyhovovalo výsledku vykreslení.

Pro každou plošku je vypočítaná intenzita osvětlení, která na plošku dopadá ze všech plošek, a která je ploškou vyřazena do prostoru, ať už odrazem, nebo pokud sama intenzitu vyřazuje.



Obrázek 17: Schéma vypočtu radiosity

Protože by na hraně mezi sousedními ploškami vznikala viditelná hrana, tak se u vrcholů, které nejsou na hraně vrcholu žádného objektu, zprůměruje intenzita všech přiléhajících plošek. Nakonec můžeme výsledek vykreslit pomocí Gouradova stínovaní.

3.2.4 Objemové algoritmy

Algoritmy, zobrazující povrchy Algoritmy, zobrazující povrchy, řadíme mezi nepřímé algoritmy. Ze vstupních dat se spočítá pomocná geometrie, která se pak dále zobrazuje a s původními daty se dál nepočítá. Vytvořená pomocná geometrie je povrch, jenž aproxi-

muje povrch původních dat. To také znamená, že jsme se přesunuli z objemových těles k hraničním tělesům. A můžeme výsledek vykreslit, například pomocí technik uvedených výše.

Přímé objemové algoritmy Naproti tomu jsou přímé zobrazovací objemové algoritmy, které se snaží zobrazovat vstupní objemová data přímo. Mezi takovéto algoritmy řadíme například raytracer POV-Ray.

4 Implementace

Tato kapitola se zaobírá už přímou implementaci vybraných algoritmu pro zobrazování konečných prvků. Nalezneme zde popis QT knihovny, stručný pohled na hlavní kostru programu a v neposlední řadě popis algoritmu pro dekompozici (simple, tree, springs), vyhlazování, shadery a úpravy grafického uživatelského rozhraní.

Aplikace je napsána pomocí QT, OpenGL a Visual Studia 2010. Na visual studio 2010 se přešlo během vývoje z Visual studia 2008.

4.1 Qt

Grafické uživatelské rozhraní (GUI) je napsáno v knihovně QT. QT je jedna z nejpopulárnějších multiplatformních knihoven pro Windows, Linux a Mac. Byla vytvořena v roce 1999 společnosti Trolltech. A v roce 2008 prodána společnosti Nokia.

Qt knihovna byla napsána pro jazyk c++, ale existují verze i pro jiné programovací jazyky, například pro python,... . Mezi její přednosti kromě multiplatformosti řadíme i dobrou dokumentaci. Pro více informací o Qt doporučuji přečíst [8],[9],[10],[11] a [12].

Aplikace byla napsána pro QT verzi 4.7.2.

4.2 Struktura aplikace

Celá aplikace se skládá z dílčích projektů.

- Projekt Database slouží k nahrávání modelu z databáze.
- Projekt DataCore obsahuje základní prvky SceneData, SceneTree, parentNode a entityNode, které slouží k obecnému vykreslení objektu pomocí hierarchie stromu. Také zde nalezneme podporu shader programu a speciální entitu pro řezaní modelu pomocí cliping plane.
- Projekt FemDataCore rozšiřuje třídy parentNode a enitityNode o potřebné elementy pro vykreslování našich konkrétních modelu. Nalezneme zde již zmíněné rozšíření `entity_Mesh` a `entity_Meshpart`, samotné data modelu v třídách Mesh a MeshPart a třídy pro výpočet a provedení rozpadu těles.
- Projekt FemManager hlavní projekt, jenž se stará o manager. A obsahuje funkci main.
- Projekt Renderer projekt, jenž obsahuje třídu SimpleViewer, která se stará o vykreslovací okno OpenGL.
- Projekt QGLViewer projekt, jenž upravuje QGLviewer pro potřeby aplikace
- Projekt GGradientViewer obsahuje výchozí třídu pro SimpleViewer.

4.2.1 Popis struktury

Hlavním projektem je FemManager, ve kterém se nachází třída Manager. Třída Manager, vycházející z třídy QMainWindow, má na starost hlavní okno, signály pro manipulaci s nahráváním objektu, gui prvky a také drží důležitou strukturu SceneData.

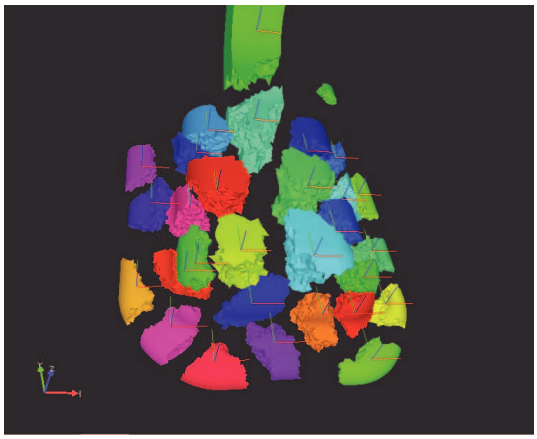
Třída SceneData, jak už z názvu vyplývá, zapouzdruje data aplikace. Je navržena jako pseudo singleton. Také uchovává stavy pro určení, jakým způsobem se má daný model vykreslit, a nastavení globálního vykreslení. Toto nastavení nalezneme v třídě DrawModelFlags. Globální nastavení se skládá z barvy pozadí, hran, bodů, velikostí alfa koeficientu, pro průhledné vykreslování. Samotné data jsou obsazena v třídě SceneTree.

Stromem, který drží částí modelu, je SceneTree. Obsahuje ukazatel na kořen stromu, částí vykreslovaného modelu a LogicInterface, který udává způsob rozkladu. Uzly stromu jsou tvořeny třídou SceneTreeNode. Z této třídy se odvozuji parentNode, jenž představuje vnitřní uzly stromu a entityNode, který představuje listy stromu. Z parentNode se odvozuje Entity_Mesh, který představuje celý model, a z EntityNode se odvozuje Entity_meshpart, jenž představuje jednotlivé domény modelu.

4.3 Implementované algoritmy

4.3.1 Dekompozice

Třída LogicInterface Jedná se o virtuální třídu, z níž se dědí třídy pro popis rozkladu modelu. I když je v názvu slovo interface, tak se nejedná o interface, jaký je například k nalezení v jave. Jedná se o abstraktní datový typ, který pouze určuje jaké hlavní funkce je třeba implementovat pro výpočet rozkladu.



Obrázek 18: Ukázka dekompozice

Třída je navržena jako návrhový vzor Abstraktní továrna. Třída úzce spolupracuje s jedním GUI prvkem. Tento prvek je QTreeWidget a zajišťuje zobrazení logické struktury rozkladu pro snadnější manipulaci (vybíraní klíčových domén/skupin pro rozpad).

QTreeWidget není nic jiného než stromová tabulka s třemi sloupci, a potomci třídy LogicInterface musí v sobě implementovat dvě funkce pro práci s QTreeWidget.

Mezi hlavní funkce patří:

- UpdateMatrix() - Funkce pro výpočet nových pozic při manipulaci s modelem pomocí rozkladu. Tato funkce se volá při manipulaci s modelem pomocí rozkladu.
- LoadPoint() - Tato funkce naplňuje třídu pro rozklad daty. Tato funkce je volaná pokaždé při změně typu rozkladu.
- CreateLogic() - nejdůležitější funkce, jenž má za úkol vytvořit logiku pro výpočet rozkladu. Tato funkce je volaná pokaždé po funkci LoadPoint()
- FillTable() - Funkce pro GUI naplňuje tabulku s daty pro manipulaci s částmi objektu. Tato funkce je volaná pokaždé po funkci CreateLogic()
- ClickTable() - Zpracování událostí při kliknutí na nějaký item v tabulce. Funkce se volá pokaždé při interakci s GUI pro rozklad.
- Reset() - Funkce pro obnovení výchozí pozice modelu. Je volaná po stisknutí příslušného tlačítka v menu, nebo při zapnutí ořezové roviny.

Třída dále obsahuje seznam bodů, který se vyplňuje pomocí funkce LoadPoint. Tento seznam bodů reprezentuje seznam domén v modelu. Pro jednoduchost jsme se omezili pouze na centra domén a vypustili jsme informaci o prostorové velikosti domény. Dále zde nalezneme seznam matic, jenž reprezentují transformaci jednotlivých částí modelu z rozkladu, a seznam elementů, které představují vnitřní logiku daného rozkladu.

Simple Jedná se o nejjednodušší způsob rozkladu. Pro svou jednoduchost nepotřebuje funkci CreateLogic. Proto tato funkce zůstává prázdná. Důležitou částí je ale funkce pro výpočet aktuální pozice domén:

```
void LISimple::UpdateMatrix(float s,float x,float y,float z)
{
    matice.clear();
    glPushMatrix();
    for(int i= 0; i < seznamElementu.size(); i++)
    {
        glLoadIdentity();
        cLIElement *u = seznamElementu.at(i);
        if(u->bRozpadUse)
            glTranslated(u->centrum.x*x + u->centrum.x*s ,u->centrum.y*y + u->centrum.y*s
                , u->centrum.z*z + u->centrum.z*s);
        MATRIX4x4 mat;
        glGetFloatv(GL_MODELVIEW_MATRIX, mat.m);

        if(matice.size() > i)
            matice[i] = mat;
        else
            matice.push_back(mat);
```

```
    }
    glPopMatrix();
}
```

Výpis 1: Algoritmus pro rozklad Simple (UpdateMatrix)

Ve funkci nejprve vyčistíme seznam matic. Pro zachování zobrazovací matice ji uložíme. Na 5. řádku projdeme v cyklu přes všechny elementy (domény) a vytvoříme matici s jednoduchým posunem závisejícím na výchozí pozici modelu a síly posunu rozkladu. Tato matice je poté vložena do seznamu matic. Nakonec vytáhneme zobrazovací matici zpět z paměti, aby bylo vše v pořádku.

Tree Rozšířený algoritmus pro dílčí ovládání modelu. Přesný popis algoritmu je napsán v kapitole 2.2.2.

```
void LIStrom::CreateLogic()
 {
    vybrany = NULL;

    vector<cUzelv2*> base;
    for(int i=0; i<seznamElementu.size(); i++)
        base.push_back((cUzelv2 *)seznamElementu[i]);

    vector<cUzelv2*>* uroven0 = CallUroven(&base,0.003);
    vector<cUzelv2*>* uroven1 = CallUroven(uroven0,0.003);
    koren = new cUzelv2Rodic();

    cVector c(0,0,0);
    for(int i=0; i < uroven1->size(); i++)
    {
        koren->seznam.push_back(uroven1->at(i));
        cUzelv2 *pom = uroven1->at(i);
        cVector vecPom(pom->centrum.x, pom->centrum.y, pom->centrum.z);

        c += vecPom;
    }
    c = c* (1.0/(double)uroven1->size());
    koren->centrum = c;

    UpdateRodic(koren);
}
```

Výpis 2: Pseudo algoritmus pro rozklad Tree (CreateLogic)

Seznam bodu (center domén) se nejdříve překopíruje do pomocného seznamu base. Následně je pomocí funkce CallUroven nad tímto seznamem vytvořen seznam shluků (uroven0). A poté je i nad seznamem shluků (uroven0) vytvořen seznam shluků (uroven1). Jako předposlední akce je vytvoření hlavního uzlu a vyplnění jeho seznamu potomků seznamem uroven1. Takto jsme získali stromovou strukturu s hloubkou 3, hlavním uzlem (kořen) a listy stromu reprezentující centra domén modelu. Nakonec je zavolaná funkce UpdateRodic, která všem uzlům nastaví správného rodiče, pro snazší průchod stromu.

```
vector<cUzelv2*> *LIStrom::CallUroven(vector<cUzelv2*>* sezUzlu,float f)
 {
 vector<cUzelv2*> *ret = new vector<cUzelv2*>();
 NajdiShlukyRodic(f/4.0,ret,sezUzlu,3);
 NajdiShlukyRodic(f,ret,sezUzlu,0);
 return ret;
 }
```

Výpis 3: Pseudo algoritmus pro rozklad Tree (CallUroven)

Funkce CallUroven je tvořena hlavně funkcí NajdiShlukyRodic(). Úlohou této funkce je najít nejdříve malé shluky a poté vetší shluky. Nakonec je celý seznam vrácen.

Následující výpis funkce NajdiShlukyRodic() byla kvůli své velikosti zredukována jen na podstatné částí a zredukované částí byly nahrazeny třemi tečkami a komentářem o funkci, kterou plnily.

```
void LIStrom::NajdiShlukyRodic(double d,vector<cUzelv2*>* sezU,vector<cUzelv2*>* sezL,
     int meze)
 {
    vector<int> * maxSez = new vector<int>;
    for(int i=0; i < sezL->size(); i++)
    {
       ...
      //vytvoren cVector a reprezentujici shluk z i-teho prvku sezname sezL

       for(int mm=0; mm < sezL->size(); mm++)
       {
          ...
           //vypocet vzdalenostÍ delka mm-teho prvku z seznamu sezL k reprezentantu
           if(delka <= d)
           {
              sez->push_back(mm);
              pocet++;
           }
       }

    ...
        //Pokud nalezeny seznam splnuje pozadavky na minimalni hodnotu a je nejvetsi nalezeny
            stava se maxSez, jinak je smazan
    }

    ...
    //Vytvoreni uzlu predstavujici rodice nalezeneho shluku
    ...
    //Vpocet stredu shluku z aritmetickeho prumemeru vsech prvku z seznamu maxSez
    ...
    //Vyrazeni nalezenych prvku z seznamu sezL

    NajdiShlukyRodic(d, sezU, sezL, meze);
}
```

Výpis 4: Pseudo algoritmus pro rozklad Tree (NajdiShlukyRodic)

Funkce NajdiShlukyRodic nejprve vyhledá v cyklu největší možný shluk. Následně pro takovýto shluk vytvoří rodiče a nalezené prvky do něj přiřadí. Nakonec jsou nalezené prvky ze seznamu vyřazeny a cyklus se rekurzivně provede znovu, dokud není seznam sezL prázdný nebo už nejde vytvořit shluky.

```
void LIStrom::UpdateMatrix(float s,float x,float y,float z)
{
    matice.clear();
    glPushMatrix();
    for(int i= 0; i < seznamElementu.size(); i++)
    {
        glLoadIdentity();
        if (((cUzelv2 *)seznamElementu[i])->setTransformation(s,x,y,z))
        {
            MATRIX4x4 mat;
            glGetFloatv(GL_MODELVIEW_MATRIX, mat.m);

            if(matice.size() > i)
                matice[i] = mat;
            else
                matice.push_back(mat);
        }
    }
    glPopMatrix();
}
```

Výpis 5: Pseudo algoritmus pro rozklad Tree (UpdateMatrix)

Ve funkci nejprve vyčistíme seznam matic. Pro zachování zobrazovací matice ji uložíme. Na 5. řádku projdeme v cyklu přes všechny elementy(domény) a vytvoříme matici s transformací podle funkce setTransformation, která se provede podle vnitřní logiky vytvořeného rozpadu. Tato matice je poté vložena do seznamu matic. Nakonec vytáhneme zobrazovací matici zpět z paměti, aby bylo vše v pořádku.

Springs Teoretický popis fungovaní Springs rozpadu je napsán v kapitole 2.2.2.

Třída cLIElement, jenž reprezentuje vnitřní prvek rozpadu a částečně tak definuje chování celé třídy pro rozpad, byla rozšířena na třídu cSpring. Do třídy cSpring byl přidán seznam ukazatelů na další třídy. Seznam ukazatelů představuje logické spojení dvou domén. Při přirovnání ke grafu je to jednosměrná hrana grafu.

Dále byla navrhnuta struktura sTMPBod pro výpočet nalezení nejbližších domén. Obsahuje kromě informace, ke které doméně patří (cSpring), také pole délky tři představující nejbližší nalezené domény. Pro výpočet aktualizace transformačních matic byla přidána struktura sTMPMinBod, která obsahuje informaci o doméně a o síle, která na tuto doménu působí.

Následující kód, popisující algoritmus pro rozpad pomocí Springs, byl kvůli své velikosti rozdělen na 4 části.

```
void LISprings::CreateLogic()
{
    vector<cSpring*> tmpList;
```

```
        vector<sTMPBod> listZBod;
        float plane = 0;

        for(int i=0; i < seznamElementu.size(); i++)
        {
            tmpList.push_back((cSpring *)seznamElementu.at(i));
        }

    sort(tmpList.begin(), tmpList.end(),sortFce);
```

Výpis 6: Pseudo algoritmus pro rozklad Springs (CreateLogic) 1/4

Nejprve je vytvořen pomocný vektor tmpList, do kterého je překopírován obsah seznamu center domén seznamElementu. Dále je tento seznam seřazen podle jedné z os. Pomocí standartní funkce sort je zavolána funkce sortFce, která zařizuje porovnávání. Tímto krokem připravujeme algoritmus na průchod zametací rovinou po důležitých bodech ze seznamu tmpList. Také je vytvořen seznam aktuálně zpracovávaných bodů listZBod.

```
    for(int i=0; i < tmpList.size(); i++)
    {
        cSpring * tmpBod = tmpList.at(i);
        plane = tmpBod->centrum.z;

        for(int j =0; j < listZBod.size(); j++)
        {
            sTMPBod * bod = &listZBod.at(j);

            // 1. vyradit body z seznamu
            //podminka pro maxVzdalenost
            if (bod->pocet == 3)
                if ((plane - bod->bod->centrum.z) > bod->nejblizsi[2].vzdalenost)
                {
                    // vyradit
                    bod->bod->pruziny.push_back(bod->nejblizsi[0].bod);
                    bod->bod->pruziny.push_back(bod->nejblizsi[1].bod);
                    bod->bod->pruziny.push_back(bod->nejblizsi[2].bod);

                    bod->nejblizsi[0].bod->pruziny.push_back(bod->bod);
                    bod->nejblizsi[1].bod->pruziny.push_back(bod->bod);
                    bod->nejblizsi[2].bod->pruziny.push_back(bod->bod);
                    //smazat z listu

                    listZBod.erase(listZBod.begin()+j);
                    j--;
                    continue;
                }
```

Výpis 7: Pseudo algoritmus pro rozklad Springs (CreateLogic)2/4

Tímto setříděným seznamem tmpList začneme procházet a posouvat zametací rovinu podle aktuálního prvku ze seznamu. Pro každý posun zametací roviny se projde seznam aktuálně zpracovávaných bodů listZBod. A jestliže je splněna podmínka pro vyřazení to-

hoto bodu, je bod vyřazen. Podmínky pro vyřazení jsou dvě. První podmínka: počet zaznamenaných nejbližších bodů je roven třem, a druhá, důležitější podmínka: vzdálenost roviny od bodu z aktuálně zpracovávaného seznamu listZBod je větší než vzdálenost bodu z aktuálně zpracovávaného seznamu listZBod k třetímu nejbližšímu nalezenému bodu.

Vyřazení bodu se provádí tím, že se nejprve vytvoří logické vazby mezi bodem, který se vyřazuje, a třemi nejbližšími nalezenými body. Nakonec je bod smazán ze seznamu listZbod. Podmínka je ukončena příkazem continue, protože se již dále tímto aktuálním bodem zabývat nemusíme.

```
        //2.spocitat vzdalenposti k tmp
        float vzd = sqrt( bod->bod->centrum.z* tmpBod->centrum.z +
                          bod->bod->centrum.y* tmpBod->centrum.y +
                        bod->bod->centrum.x* tmpBod->centrum.x);

        // jestlize je pocet mensi jak 3 tak pridat
        if (bod->pocet < 3)
        {
            bod->nejblizsi[bod->pocet].vzdalenost = vzd;
            bod->nejblizsi[bod->pocet].bod = tmpBod;
            bod->pocet++;
            continue;
        }

        if (vzd < bod->nejblizsi[0].vzdalenost)
        {
            bod->nejblizsi[2] = bod->nejblizsi[1];
            bod->nejblizsi[1] = bod->nejblizsi[0];

            bod->nejblizsi[0].vzdalenost = vzd;
            bod->nejblizsi[0].bod = tmpBod;
            continue;
        }

    if (vzd < bod->nejblizsi[1].vzdalenost)
    {
        bod->nejblizsi[2] = bod->nejblizsi[1];
        bod->nejblizsi[1].vzdalenost = vzd;
        bod->nejblizsi[1].bod = tmpBod;
        continue;
    }

    if (vzd < bod->nejblizsi[2].vzdalenost)
    {
        bod->nejblizsi[2].vzdalenost = vzd;
        bod->nejblizsi[2].bod = tmpBod;
        continue;
    }
}

//3. pridat tmp do seznamu
sTMPBod novy;
```

```
            novy.pocet = 0;
            novy.pozice = 0;
            novy.bod = tmpBod;
            listZBod.push_back(novy);

        }
```

Výpis 8: Pseudo algoritmus pro rozklad Springs (CreateLogic)3/4

Pokud však nebyly splněny podmínky pro odstranění, tak zjišťujeme, zda-li bod ze seznamu tmpList není mezi třemi nejbližšími body k bodu ze seznamu listZBod. Pokud se však tento bod nalézá v blízkosti, tak je zařadíme na příslušné místo do seznamu nejbližších bodů. Nakonec je bod ze seznamu tmpList zkopírován do seznamu listZbod, aby se i pro něj nalezly nejbližší body.

```
        for(int j =0; j < listZBod.size(); j++)
        {
            sTMPBod * bod = &listZBod.at(j);
            for( int i =0; i < bod->pocet; i++)
            {
                bod->bod->pruziny.push_back(bod->nejblizsi[i].bod);
                bod->nejblizsi[i].bod->pruziny.push_back(bod->bod);
            }
        }

        matice.clear();
        for(int i=0; i< seznamElementu.size();i++)
        {
            glLoadIdentity();
            MATRIX4x4 mat;
            glGetFloatv(GL_MODELVIEW_MATRIX, mat.m);
            matice.push_back(mat);
        }
    }
```

Výpis 9: Pseudo algoritmus pro rozklad Springs (CreateLogic)4/4

Po skončení cyklu procházející seznam tmpList se můžou nacházet v seznamu aktuálně zpracovávaných bodu listZbod body, pro které nebyly vytvořeny logické vazby a nebyly vyřazeny. Proto je nutné pro tyto body vytvořit logické vazby a vyřadit je. A nakonec je ještě vytvořen seznam matic pro výpočet transformací.

Funkce pro aktualizaci transformačních matic je oproti dekompozici Tree složitější. Funkce pracuje následovně: počáteční stav mějme takový, že jedna z domén byla vybrána a posunuta v prostoru nějakým směrem. Očekávejme, že díky Springs dekompozici se následně posunou i ostatní domény stejným směrem, ale jinou sílou. Síla, o kterou se domény posunou, bude záviset na jejich vzdálenosti od první vybrané posunuté domény. Čím vzdálenější budou domény, tím by měla být síla menší. Výpočet vzdálenosti zde nahradíme vazbami mezi jednotlivými doménami (pružinkami). Díky tomu ve výsledku nemusí na vzdálenosti záviset síla, ale pouze na logickém propojení mezi doménami.

```
void LISprings::UpdateMatrix(float s,float x,float y,float z)
```

```
{
    glPushMatrix();
    vector<int> pouziteUzly;
    vector<sTMPMinBod*> cekaNaZpracovani;
    cSpring * b = (cSpring *)seznamElementu.at(vybrany);
    pouziteUzly.push_back(vybrany);

    for(int i=0; i < b->pruziny.size(); i++)
    {
        sTMPMinBod * zrp = new sTMPMinBod;
        zrp->bod = b->pruziny.at(i);
        zrp->vzdalenost = 0.9;
        cekaNaZpracovani.push_back(zrp);
    }

    //Vykonat posunuti
    glLoadIdentity();
    glTranslated(b->centrum.x*x*10 ,b->centrum.y*y*10, b->centrum.z*z*10);
    MATRIX4x4 mat;
    glGetFloatv(GL_MODELVIEW_MATRIX, mat.m);
    matice[vybrany] = mat;


    for(;cekaNaZpracovani.size() != 0 ;)
    {
        sTMPMinBod* pr = cekaNaZpracovani.at(0);
        cekaNaZpracovani.erase(cekaNaZpracovani.begin());

        if(std::find(pouziteUzly.begin(), pouziteUzly.end() ,pr->bod->node->getID()) !=
            pouziteUzly.end())
            continue;
        pouziteUzly.push_back(pr->bod->node->getID());

        //Vykonat posunuti
        glLoadIdentity();
        glTranslated(b->centrum.x*x*pr->vzdalenost*10 ,b->centrum.y*y*pr->vzdalenost*10, b
            ->centrum.z*z*pr->vzdalenost*10);
        MATRIX4x4 mat2;
        glGetFloatv(GL_MODELVIEW_MATRIX, mat2.m);
        matice[pr->bod->node->getID()] = mat2;

        for(int i=0; i < pr->bod->pruziny.size(); i++)
        {
            sTMPMinBod * zrp = new sTMPMinBod;
            zrp->bod = pr->bod->pruziny.at(i);
            zrp->vzdalenost = 0.8* pr->vzdalenost;
            cekaNaZpracovani.push_back(zrp);
        }

    }

    glPopMatrix();
}
```

Výpis 10: Pseudo algoritmus pro rozklad Springs (UpdateMatrix)

Funkce UpdateMatrix si nejdříve připraví dva seznamy. Jeden seznam pouziteUzly bude představovat uzly, pro které byl již výpočet transformace uskutečněn a již dále s ním nemusíme počítat. A seznam cekaNaZpracovani, který slouží pro správnou posloupnost zpracovávání uzlu. Na pořadí, v němž se uzly budou zpracovávat, záleží. Logická struktura propojení mezi doménami je graf. Mezi primárním uzlem (ten, jenž je vybrán a posunut) a nějakým dalším uzlem, jenž se má také transformovat, existuje mnoho cest. Z těchto všech možných cest hledáme tu nejkratší. K tomu nám slouží právě seznam cekaNaZpracovani, který bude zajišťovat, v jakém pořadí se mají uzly zpracovávat.

Dále provedeme první výpočet posunutí pro vybranou primární doménu (uzel). Nejprve vezme všechny logické vazby, jdoucí od tohoto uzlu k jiným, a přidáme je do seznamu cekaNaZpracovani. Pro tyto účely využijeme strukturu sTMPMinBod, která je obohacena o konečnou sílu působící na doménu, která je na druhé straně logické vazby (pružiny). Posléze vypočteme transformační matici pro vybranou doménu a uložíme ji.

Potom v cyklu procházíme seznam cekaNaZpracovani. Pro každý cyklus vezmeme aktuální prvek ze seznamu. Projdeme seznam pouziteUzly a pokud se v tomto seznamu náš aktuální uzel nenachází, tak ho tam přidáme. Dále vypočteme pro tento uzel transformační matici a uložíme ji. Nakonec vytáhneme z aktuálního prvku seznam všech logických vazeb, které u sebe má, a uložíme je do seznamu cekaNaZpracovani.

Cyklus končí v okamžiku, kdy není žádný prvek v seznamu cekaNaZpracovani.

4.3.2 Shader - phong

Teoreticky popis Phongova stínovaní je napsán v kapitole 2.3.2.

Navržená struktura pro vykreslení modelu má v určitém pohledu jednu vlasnost, která je potřeba vyřešit před samotným vykreslením. Vnitřní stěna mezi dvěma přilehajícími doménami se duplikuje. Takto sice vzniknou dvě stěny, každá u jedné domény, ale tyto stěny sdílejí jednu normálu. Při posunutí či otočení domény může být vidět, jak se nekorektně stínují tyto stěny. Proto při vykreslování v shaderu se každá normála trojúhelníku, která by byla ve stejném směru jak je pohled kamery, otáčí. jak znázorňuje výňatek kódů z vertex shaderu.

```
if (dot(vViewVec, vNormal) < 0.0)
    vNormal *= -1.0;
```

Výpis 11: Pseudo algoritmus pro otočení normály

Zde ukázaný kód z pixel shaderu je zredukovaný jen na podstatné částí.

```
...
void main (void)
{
    ...
    float lambertTerm = dot(normalize(vViewVec), vNormal);
```

```
    if (lambertTerm > 0.0)
    {
        final_color  += color * lambertTerm;

        vec3 E = normalize(vViewVec);
        vec3 R = reflect(-normalize(vViewVec), vNormal);
        float specular = pow( max(dot(R, E), 0.0), 30);
        final_color  += MSpecular*(specular/5.0);
    }

    gl_FragColor =  final_color ;

}
```

Výpis 12: Pixel shader

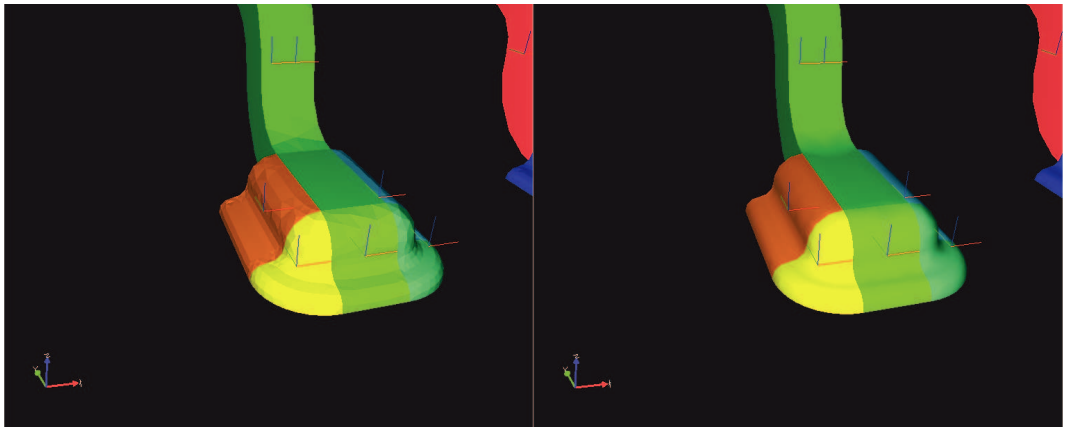
Nejprve se vypočte úhel mezi směrem kamery a normálou vykreslované stěny. Poté se rozhodne, zda je stěna osvícená. Nakonec se přidá reflexní složka a spekulární složka.

4.3.3 Vyhlazování

Popis algoritmu pro vyhlazování byl napsán v kapitole 2.3.1.

Pro potřeby algoritmu pro vyhlazování bylo potřeba zajistit správné přisazení reálných ID vrcholů jednotlivých elementů, protože se v průběhu zpracovávání modelu ID vrcholu měnily. Řešením tohoto problému bylo vytvoření speciálního pole integeru s názvem RealID. Toto pole se poté vyplnilo reálnými indexy vrcholů ve funkci InitializeMeshParts třídy FEMCreator z dat GeometryInfo.

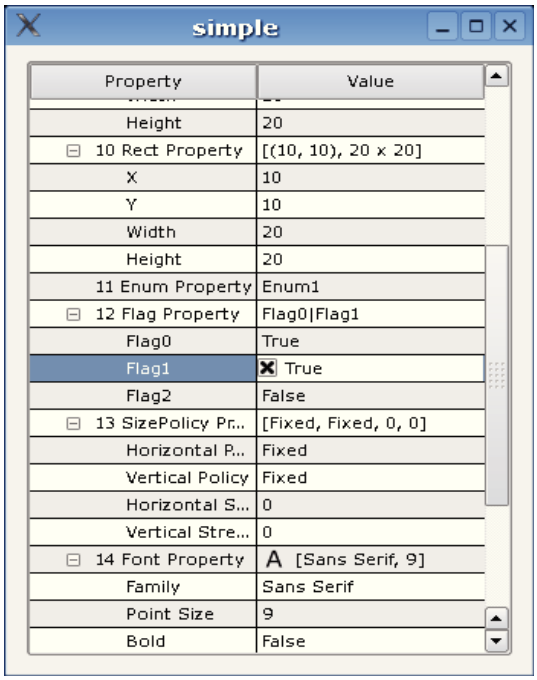
Následné vyhlazení probíhá po zavolání funkce smooth, která je funkcí třídy SceneTree. Pro zrychlení algoritmu je pole RealID setříděno. A ve funkci pomSSrovnej se provede samotné vyhlazení.



Obrázek 19: Ukázka vyhlazení

4.3.4 Vedlejší úlohy

GUI Jednou z vedlejších úloh pro GUI bylo navržení přehledného a jednoduchého systému nastavení (například pro barvu, pozadí, bodu, apod..). Pro tuto funkci bylo využito externího doplňku pro QT, přesněji šlo o třídu QtTreePropertyBrowser. Jedná se o jednouchý systém pro zobrazování nastavení formou tabulky.



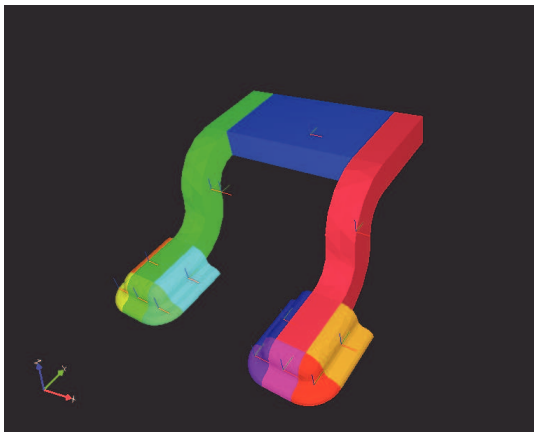
Obrázek 20: Ukázka QtTreePropertyBrowser

Použití této třídy najdeme uvnitř třídy visualDockWidget. K této třídě byla navrhnuta speciální třída visualVariantItem, která byla zděděna z QtVariantProperty a představuje jednotlivé položky ze seznamu nastavení. Interakce s tímto GUI prvkem byla zprostředkována skrze funkci valueChangedProp, která byla spojena pomocí funkce connect s slotem valueChanged.

5 Testování

Tato poslední kapitola je věnována testování aplikace. Nalezneme zde popis modelu, konfiguraci PC, na kterých byly provedeny testy. Testy byly provedeny ve třech vlastnostech. Jako vlastnosti, které budou otestovány, byly vybrány fps, vytížení paměti a zátěž CPU. U každého testu nalezneme přehledný graf a zhodnocení výsledků tohoto testu.

Testování je nedílnou součástí každé aplikace. Na této diplomové práci (aplikaci) byly prováděny testy tří vlastností. První vlastností bylo počet frame per second (snímku za sekundu), dále označováno jen jako fps. Druhou vlastností byla vytíženost CPU a třetí vlastností bylo sledování paměti RAM.



Obrázek 21: Ukázka model 1

Aplikace byla testovaná na počítači s konfiguraci Intel Core2 Duo CPU 2.10 GHz, NVIDIA GeForce G205M, 4 GB RAM a systém Windows 7.

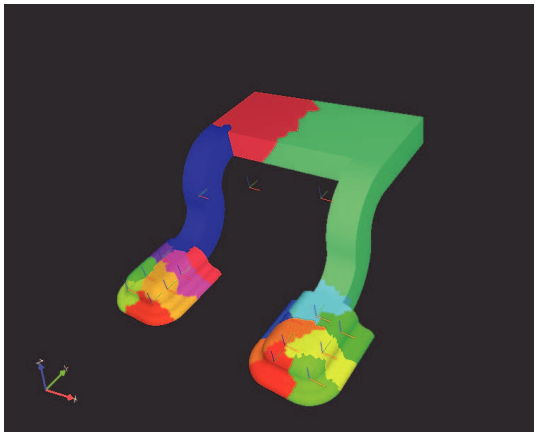
Jako testovací modely byly zvoleny 3 základní modely, jenž se načítaly z disku a byly ve speciálním formátu. Modely vypadají totožně, jenom se liší v detailnosti propracování a počtu domén.

První model zabírá na disku 1,94 MB. Má 3045 vrcholů a skládá se z 13 domén. Na obrázku 21 si můžete prohlédnout, jak model vypadá.

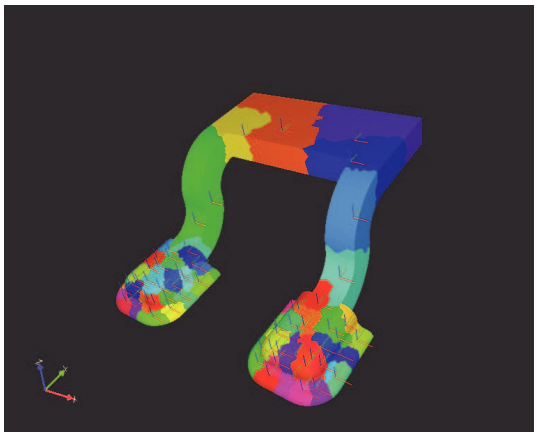
Druhý model zabírá na disku 15,7 MB. Má 90081 vrcholů a skládá se ze 17 domén. Na obrázku 22 si můžete prohlédnout, jak model vypadá.

Třetí a nejsložitější model ze všech testovaných modelů zabírá na disku 52,6 MB. Má 273976 vrcholů a skládá se z 64 domén. Na obrázku 23 si můžete prohlédnout, jak model vypadá.

K testování fps byly použity údaje, které poskytuje sama aplikace, pro testovací účely byla vypnuta vertikální synchronizace. Takže rychlost fps neodpovídá skutečné rychlosti zobrazování na monitoru, ale udává potenciální rychlost výpočtu pro zobrazení na monitoru. Na testování vytíženosti CPU byl použít program Samurize, jenž je k dispozici pod licencí Freeware. Autorem tohoto programu je Gustav & Oscar Lundh. K monitorování paměti RAM byl použít program RAM graf verze 1.999, který je taky k dispozici pod licencí Freeware. Autorem programu RAM Graf je DExUS.



Obrázek 22: Ukázka model 2



Obrázek 23: Ukázka model 3

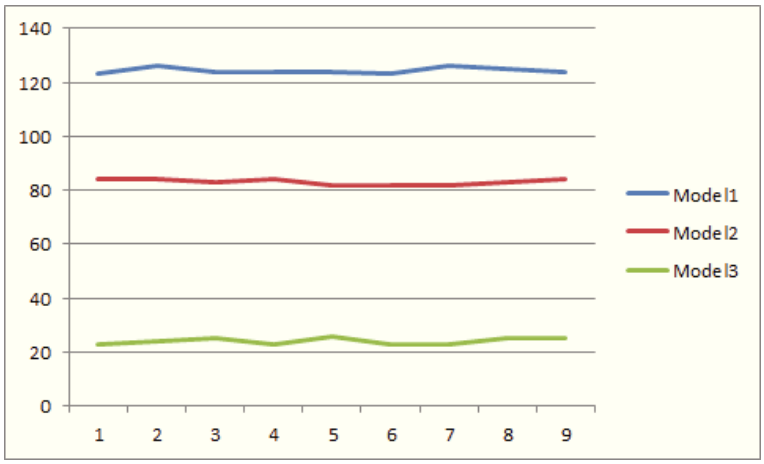
FPS tedy počet snímku za sekundu je, jak už název vypovídá, důležitý pro rychlost vykreslování. Rychlost vykreslovaní je jedním z nejdůležitějších faktoru pro počítačovou grafiku. Nízká hodnota fps může způsobovat sekání obrazu nebo kazit iluzi pohybu. Pro bezproblémové sledovaní pohybu objektu pro naše oko je nejmenší udávanou hodnotou 25 fps. Naprosto dostačující hodnota bývá 60 fps.

Výkon CPU je taktéž důležitý. Větší výkon CPU znamená větší námahu na počítač a někdy znamená i pořízení rychlejšího CPU a tedy dražší náklady.

Velikost spotřebovávané paměti by měla jít ruku v ruce s velikostí načítaného objektu. Takže i zde by měla být snaha o co nejmenší spotřebu paměti, protože velká spotřeba paměti znamená větší paměť, a to zase navyšuje potřebu větší investice peněz.

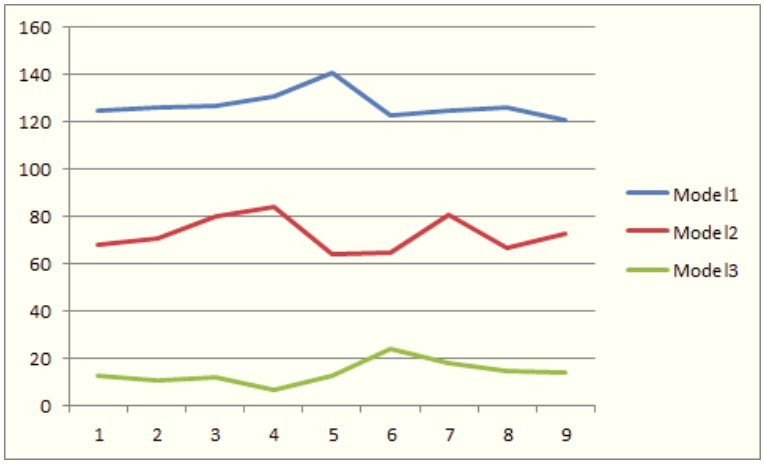
5.1 Testování - fps

Testování fps probíhalo ve třech krocích. Prvním krokem bylo změření fps v klidovém stavu, kdy se s modelem nijak nemanipulovalo. Druhý test se zaměřil na měření fps při náhodné manipulaci s modelem. Ve třetím testu bylo měřeno fps při manipulaci s částečně průhledným modelem. Testy probíhaly nad shaderem typu phong.



Obrázek 24: Testování fps - stav v klidu

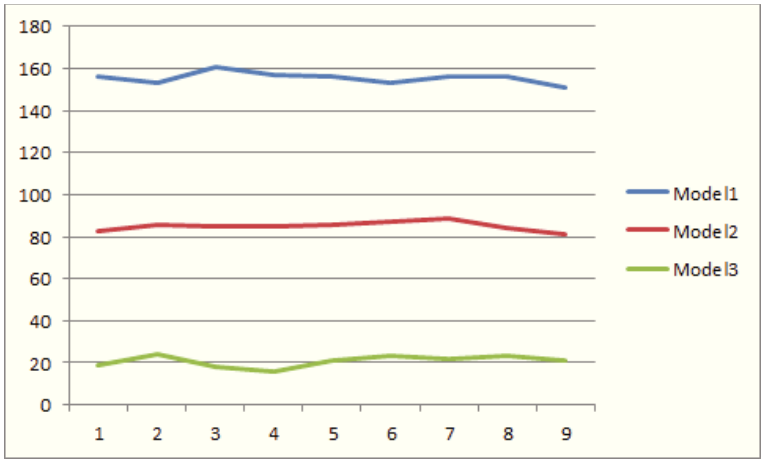
V grafu vidíme, že model 1 a model 2 je ještě nad hranicí dobré rychlosti fps. Model 3 je však už pod hranicí, počet fps se pohybuje okolo 24 snímků za sekundu, což je přibližně frekvence snímků u filmu, ale u počítačové grafiky je to už na hraně se sekáním obrazu. Model 3 má však 91krát více vrcholů než model 1 a vzhledem k tomu je 24 fps dostačující.



Obrázek 25: Testování fps -pracovní stav

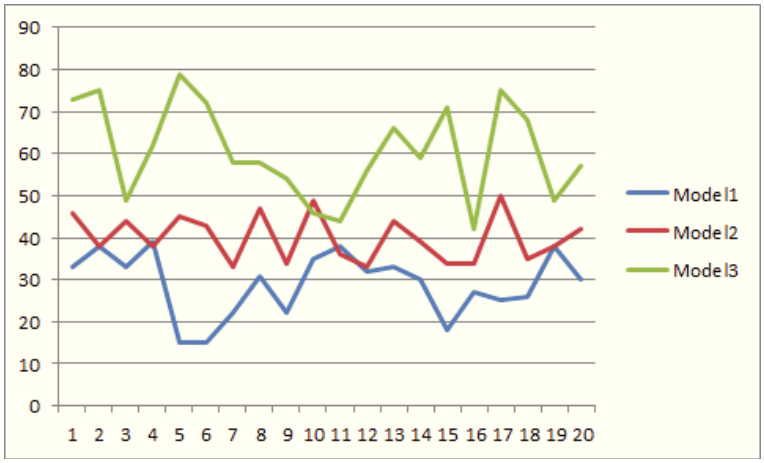
Na grafu 2 vidíme mírný propad fps při manipulaci s modelem. To je pochopitelné a očekavatelné, protože při manipulaci s objektem se vykreslují navíc další grafické prvky,

jako jsou zvýrazněné hrany ploch nebo zvýrazněné body, také samotná manipulace potřebuje určitý výpočet. Mírný propad vidíme u všech třech modelů. Tento propad se pohybuje od 1% do 10%.



Obrázek 26: Testování fps s průhledností

Na grafu 3 lze vidět překvapující zvýšené fps oproti normálu. Očekávali bychom propad fps, protože vykreslování s průhledností je složitější. Avšak průhlednost je použita na objekty, u kterých je vypnuta manipulace, a dále jsou tyto modely zjednodušeně vykreslovány. Proto je výsledné vykreslení obrazovky u částečně průhledných modelů rychlejší než normální vykreslení.



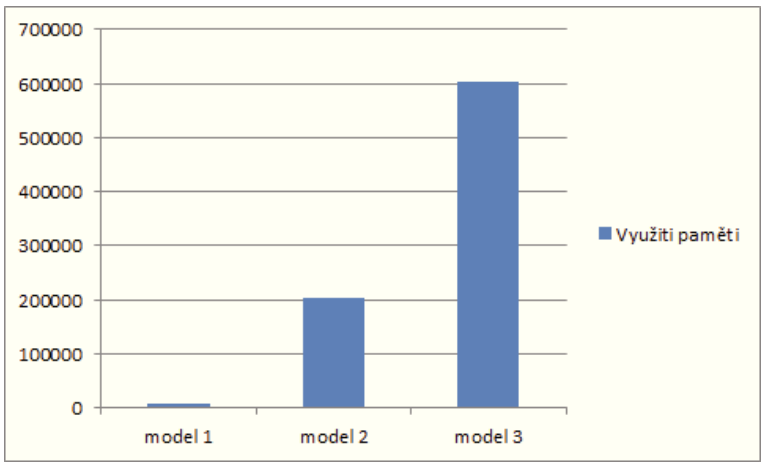
Obrázek 27: Testování CPU

5.2 Testování - CPU

Testováním vytíženosti CPU (obrázek 27) bylo zjištěno, že nejsložitější model 3, jenž má 91krát více vrcholů než model 1, zatěžuje CPU dvakrát tak více než model 1.

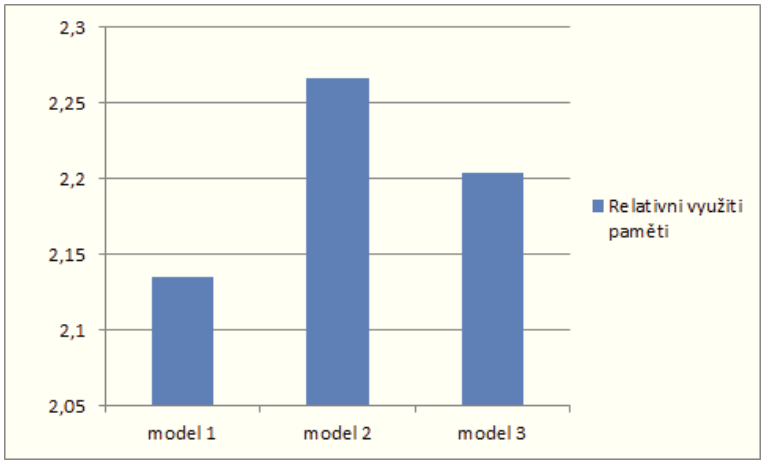
5.3 Testování - Pamět

Při testu paměti bylo hlavním cílem zjistit její využití před a po načtení jednotlivých modelu. Testy byly prováděny dva. Jeden na zjištění, kolik daný model zabírá v paměti místa. A druhý test, jenž byl vypočten z prvního, byl zaměřen na zjištění poměru využití paměti modelem k počtu vrcholů, ze kterých se model skládá.



Obrázek 28: Testování paměti

Graf 28 nám ukazuje absolutní využití paměti. Tento graf nám však neřekne skoro nic, proto je lepší podívat se na další graf 29, kde je zobrazen poměr využití paměti k počtu vrcholů modelu. Zde vidíme, že se hodnota pohybuje okolo čísla dvě. Z toho vyplývá, že růst velikosti modelu v paměti je lineární k velikosti modelu.



Obrázek 29: Testování paměti/velikost modelu

6 Závěr

Všechny cíle byly splněny. Po implementační stránce byly navrhnuty a vypracovány všechny zadané úkoly, jmenovitě dekompozice Simple,Tree,Springs, vyhlazování, shader, úprava a přidání GUI rozhraní a přidání alfa kanálu. Testy na fps prokázaly, že si aplikace poradí s malými a středně velkými modely. U velkých modelů testy prokázaly, že zobrazovací rychlost fps je na hraně přípustné úrovně, ale stále dostačující. Testy na pamět nám ukázaly, že spotřebovaná pamět roste linearně s velikostí modelů. Velkou roli pro rychlost fps také přispělo permanetní nastavení kreslení pouze vnějších elementů. Doposud se kreslily i vnitřní elementy.

Implementace dekompozice do aplikace se zdařila. Všechny tři dekompozice mají navrhnuté vlastnosti a schopnosti. Dekompozice Simple je jednoduchou alternativou, která je vhodná pro rychlou dekompozici pro jednoduché modely. Dekompozice Tree je vhodnou volbou pro složitější modely, jenž potřebují při rozpadu částečnou kontrolu nad modelem. Poslední třetí dekompozice Springs se ukázala jako vhodná volba pro složité modely, pro které jsou predešlé dvě dekompozice nedostačující.

Výpočet vyhlazení modelu je rychlý a normály ve vrcholem správně nastavují. Vyhlazování a navrhnutý shader zlepšuje obrazový vjem modelu. Přidané prvky grafického uživatelského rozhraní byly řádné umístěny do oblasti, kam by měly intuitivně patřit, taktéž jejich ovládání je jednoduché a přehledné.

Vykonané testy ukázaly dobrou kvalitu aplikace. Testy na fps prokázaly dostatečně rychle vykreslení i pro složité modely. Testy na využití paměti prověřily dobré hospodaření s paměti, tedy využití paměti roste lineárně s velikostí modelu. A CPU testy nezaznamenaly, žádné velké zatížení procesoru. Celkové je aplikace svižná a rychlá.

7 Reference

[1] J. Žára, B. Beneš, J. Sochor, P. Felkel *Moderní počítačová grafika*, Computer Press Brno, 2004, druhé vydání.

[2] Tim Foley, Pat Hanrahan: *Spark: modular, composable shaders for graphics hardware*, ACM Trans. Graph., 2011.

[3] Ed Angel, Dave Shreiner: *Teaching a Shader-Based Introduction to Computer Graphics*, IEEE Computer Graphics and Applications, 2011.

[4] Mike Bailey: *Using GPU Shaders for Visualization*, IEEE Computer Graphics and Applications, 2009.

[5] Yinlong Sun, Qiqi Wang: *Interference Shaders of Thin Films*, Comput. Graph. Forum, 2008.

[6] Mike Bailey: *Teaching OpenGL shaders: Hands-on, interactive, and immediate feedback*, Computers and Graphics, 2007.

[7] Wolfgang Engel: *GPU Pro: Advanced Rendering Techniques*, ISBN-10: 1568814720, 2010.

[8] Alan Ezust, Paul Ezust: *Introduction to Design Patterns in C++ with Qt (2nd Edition)*, Prentice Hall, 2008.

[9] Jasmin Blanchette and Mark Summerfield: *C++ GUI Programming with Qt 4 (2nd Edition) - The official C++/Qt book*, Prentice Hall, 2008.

[10] Mark Summerfield: Advanced Qt Programming: *Creating Great Software with C++ and Qt 4*, Prentice Hall, 2010.

[11] Johan Thelin: *Foundations of Qt Development*, Apress, 2007.

[12] Frank H. P. Fitzek, Tommi Mikkonen, Tony Torp: *Qt for Symbian*, Forum Nokia, 2010.

[13] Boudewijn Rempt: *Python: Qt Edition*, OpenDocs, LLC , 2002.

[14] J. N. Reddy: *Introduction to the Finite Element Method*, McGraw-Hill Science/Engineering/Math; 2 edition, 1993.

[15] David Moratal: *Finite Element Analysis*, Sciyo, August, 2010

[16] S.S. Bhavikatti: *Finite Element Analysis*, New Age International, 2007.

[17] Daryl L. Logan: *A First Course in the Finite Element Method*, Cengage Learning, 2007

[18] Thomas J. R. Hughes: *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis (Dover Civil and Mechanical Engineering)*, Dover Publications, 2000.

[19] Tirupathi R. Chandrupatla, Ashok D. Belegundu: *Introduction to finite elements in engineering*, Prentice Hall, 1997.

[20] Eduard Sojka, *Počítačová grafika II: metody a nástroje zobrazování 3D scén*, VŠB-TUO.

[21] Andrew S. Glassner: *An Introduction to Ray Tracing*, Morgan Kaufmann, 1989.

[22] Peter Shirley, R. Keith Morley: *Realistic Ray Tracing*, A K Peters Ltd, 2009.

[23] Tomas Möller, Eric Haines, Naty Hoffman: *Real-Time Rendering*, CRC Press, 2008.

[24] Matt Pharr, Greg Humphreys, Greg Humphreys (Ph. D.): *Physically Based Rendering: From Theory To Implementation*, Morgan Kaufmann, 2004.

[25] Henrik Wann Jensen: *Realistic image synthesis using photon mapping*, A K Peters, 2001.

[26] John R. Wallace: *Radiosity and Realistic Image Synthesis*, Morgan Kaufmann, 1993.

[27] Arie Kaufman: *Volume visualization*, IEEE Computer Society Press, 1991.

[28] Min Chen, Arie Kaufman, Roni Yagel: *Volume graphics*, Springer, 2000.

[29] Erhard Neher, Alistair Savage, Weiqiang Wang: *Geometric Representation Theory and Extended Affine Lie Algebras*, American Mathematical Soc., 2011.

[30] Michael Richard Gosz: *Finite Element Method: Applications in Solids, Structures, And Heat Transfer*, Taylor and Francis, 2006.

[31] Young W. Kwon, Hyochoong Bang: *The Finite Element Method Using Matlab*, CRC Press, 2000.